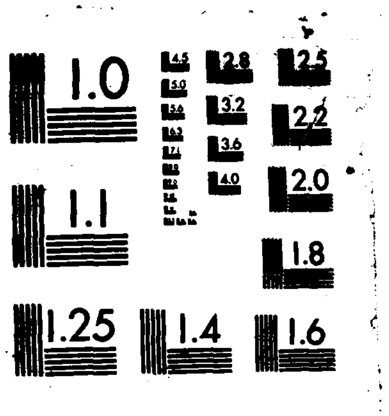
AD-A182 460 GENERATING INCREMENTAL VLSI COMPACTION SPACING CONSTRAINTS(U) STANFORD UNIV CA C W CARPENTER ET AL 1986 MDA903-83-C-0335 1/1

UNCLASSIFIED F/G 9/1 NL

END
8-87
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(1) dac 87

Generating Incremental VLSI Compaction Spacing Constraints

Clyde W. Carpenter & Mark Horowitz
Stanford University, Stanford CA 94305

1986

Abstract

This paper describes using adjacency lists to incrementally generate design rule spacing constraints. The algorithm generates the smallest complete set of constraints for a design, yielding fast compaction, and is as fast or faster than ordinary constraint generation methods even when the incremental features are not used. The adjacency list data structure allows one to very quickly move, insert or delete objects and generate an updated set of constraints.

Introduction

Compaction is the translation from a high level description of a circuit down to the detailed layout needed for fabrication, trying to make as compact a layout as possible without violating any design rules. Working at a more abstract level offers many advantages, including freeing the designer from worrying about the design rules and making it easier to create new masks when the rules change.

One problem with using a compactor is the long run times needed for large designs. If the resulting layout is too big, some of the cells have to be changed and the whole circuit recompacted. This design loop time could be drastically reduced by an incremental compactor that allowed one to edit the schematic and directly see the updated layout.

An incremental compactor needs to incrementally generate design rule constraints. Rules can be divided into two classes. The first class holds things together, keeping the parts of transistors and contacts aligned and wires connected to their endpoints. The set of constraints needed to enforce these rules is invariant during compaction, it is created once using a database describing objects in the current technology.

The second class of design rules are the spacing rules. They provide the margins needed by the fabrication process to keep adjacent objects in the layout from interfering with each other. The set of constraints needed to enforce these rules depend on the x and y coordinates of the objects. The compaction process is usually divided into two one-dimensional problems because of the complexity of true two-dimensional optimization. Performing a one-dimensional compaction step changes the set of constraints needed by the other dimension. Moving a layout level rectangle, a tile, up or down in the y direction can change the set of tiles on the its left and right, thus changing the set of constraints needed for compaction in the x direction. Throughout this paper we will describe the algorithms for creating and updating the set of x direction constraints affected by y direction movements. The y-x case is symmetrical.

After discussing previous methods used to generate constraints, this paper describes the new data structure and algorithms used to generate and update the spacing constraints. We start with a simplified model of the problem, single layer designs, and then describe the extentions for more general situations.

Background

Many algorithms have been used to generate the spacing constraints. The simplest one just checks for interactions between every pair of tiles. The distance between two tiles is taken to be the maximum of the x and y distances between their edges. Thus an x direction constraint between two tiles is needed only if they are seperated by less than the minimum legal spacing in the y direction. This algorithm is easy to code but has the drawback that although each comparison is quick, n^2 comparisons are required.

The number of comparisons can be reduced by first sorting the tiles by their bottom coordinates and then comparing each tile with just the tiles in the horizontal band above it. This gives $O(n^{1.5})$ total comparisons for a roughly square layout[1]. These algorithms have another disadvantage, for normal layouts they produce $O(n^{1.5})$ constraints when, because of transitivity, only $O(n)$ constraints are needed.

A more complex algorithm uses Shadowing. The tiles are lexicographically sorted on the (x, y) coordinates of their lower-left-hand corner. A vertical frontier forms and moves to the right as the tiles are processed. The frontier

This material is based upon work supported under a National Science Foundation Graduate Fellowship. This work was supported in part by the National Science Foundation under Grant DMC8451822, and in part by the Defense Advanced Research Projects Agency under contract MDA 903-83-C-0335.

consists of tiles that could possibly be seen by a tile to the frontier's right. Constraints are generated from tile(s) in the frontier to each tile in turn before that tile is added to and shadows part of the frontier. A tile is shadowed when constraints from it to any possible tile to the right of the frontier are superfluous. This is an elaboration of the basic algorithm used in many design rule checkers[2]. It can be difficult maintaining this frontier since a tile often shadows just part of another tile and a large tile can be cut into pieces by the shadows of several small tiles.

Another idea is to use Intervening Groups. Tiles that are rigidly held together are grouped into features. As all the pairs of features are compared an approximation of the longest path between every pair is created. Constraints from previously compared features may already require two features to be greater than the maximum design rule distance apart saving the expense of comparing all the tiles in them. One method keeps track of just one longest path[3] resulting in extra comparisons. Another uses a limited depth search and a square bitmap[4] to store previous results, requiring a great deal of memory. While these are still $O(n^2)$ algorithms, they are very fast $O(n^2)$ algorithms.

The above algorithms need to be completely rerun whenever anything is moved or changed. One way to reduce these recomputations to use the wires to divide the layout into regions[5]. The set of tiles in or on the edge of each region is invariant during compaction because the wires remain connected to their endpoints. An every-pair algorithm can be used separately on each region, when tiles are moved only the regions affected need to be redone. This algorithm runs slowly with large regions.

The corner-stitching[6] data structure allows more general quick changes. While designed for a layout editor, the routines used to check for design rule errors are similar to the ones needed to generate the design rule constraints. Corner-stitching uses space tiles in addition to the other types of layout tiles in order to completely cover the plane. The tiles are kept in a canonic form of maximal horizontal strips by cutting tiles horizontally and recombining them giving preference to width over height. This canonic form is useful in a layout editor since it prevents fragmentation, but it has some drawbacks for compaction: it loses x-y symmetry and obscures the mapping between higher level descriptions and the layout.

A variant of corner-stitching combined with shadowing forms the basis for our algorithm using adjacency lists. The algorithm retains the speed of shadowing and the incremental properties of corner-stitching but is tailored to generating design rule constraints.

Adjacency Lists

This section describes the adjacency lists data structure. Each tile has a list of the tiles adjacent to its left edge and another list for its right edge. Two tiles are adjacent if and only if a spacing constraint is needed to keep them apart. The relationship is symmetric; if A is left adjacent to B (in B's left list) then B is right adjacent to A. The adjacency information is stored using threaded lists, each tile record has left, right, up and down pointers. The lists are threaded clockwise, the left (right) pointer points to the lowest left (highest right) adjacent tile and the up (down) pointers are followed for the rest of the left (right) adjacency list (see Figure 1A). The lists are not always nil ended, it is possible for a tile to be the last tile in one tile's adjacency list and the first tile in another's (see dark tile in Figure 1B). But the down (up) pointer of a record is always part of the right (left) adjacency list of the tile pointed to by that record's left (right) pointer.

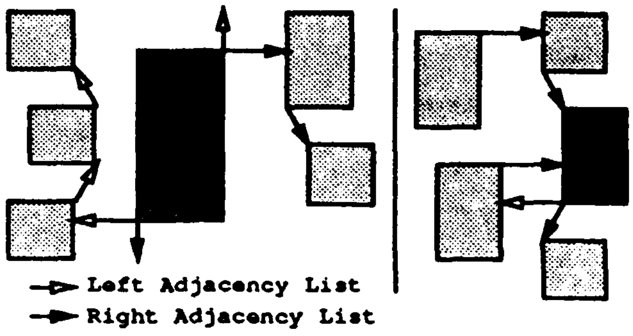


Figure 1: A. Pointers B. Non-Nil Ended List

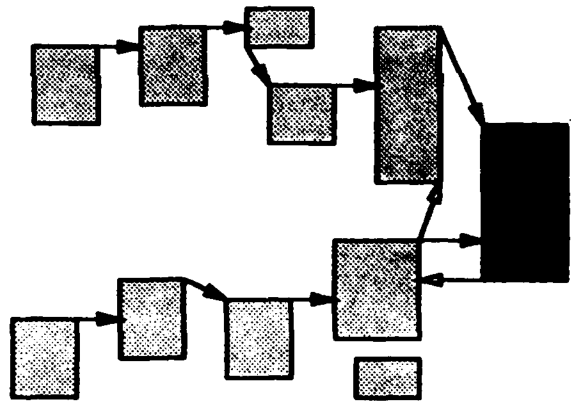


Figure 2: A Concave Left Turnaround

The algorithms often search for a turnaround point while transversing the adjacency lists. A turnaround is a group of tiles with an empty concave space on their left or right. The lower half of these tiles have nil up pointers and the upper half have nil down pointers. In Figure 2 the rightmost tile most be found to travel up or down within the turnaround. Right pointers tend to go upwards since they point to top right adjacencies. So to find a concave left turnaround from below, one just follows right pointers until a non-nil up pointer is found. To find one from above, the right and down pointers are followed to find bottom right adjacencies. The turnaround tile is the first tile whose left pointer does not point back to the previous bottom right tile.

The tiles below the turnaround can become adjacent to tiles whose bottoms move down into the turnaround's empty concave space while the tiles above are looked at when tops move up.

Single Color Case

This section describes the algorithms used to maintain the adjacency lists for the simple case where there is only one color (type) of tile and the only design rule is that tiles can touch but not overlap. This is equivalent to replacing a spacing rule of k with a spacing rule of 0 after bloating all the tiles by k/2. The tiles are stored in a single plane with a frame of four special tiles to give all the other tiles left and right neighbors and to bound searches. The growing and shrinking of tiles are the basic operations used to move, delete and insert tiles, so they will be described first.

Growing Tiles

Tiles are grown by moving their tops up or bottoms down. The cases are symmetric so we will describe moving tops up. As a tile grows, its path may cut old adjacencies and cause new ones to be added to its lists. The main part of this algorithm is a loop, scaning counter-clockwise upwards to look for adjacencies that cross the path of the growing tile. The scan searches right for a turnaround. Then it goes up and then left until a tile is found to the left of the growing tile. That is, we scan around concave left turnarounds to find all the crossing adjacencies, one-by-one, working upwards until a crossing is found above the final top of the growing tile.

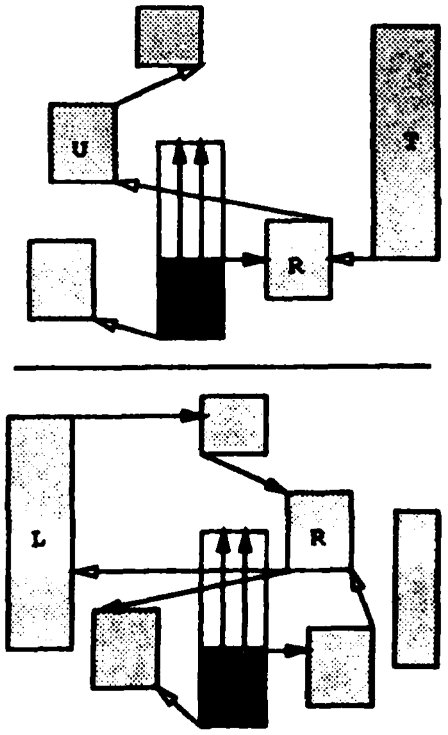


Figure 3: A. Breaking a Left List B. Breaking a Right List

Two cases can occur at the turnaround point: the up tile can be to the left or right of the growing tile. If it is to the left then the growing tile breaks into the middle of the turnaround tile's left adjacency list, removing the up tile from that list and adding it to its own left adjacency list (see Figure 3A). In the other case, the left pointers are followed to find a pair of tiles, one to the right and one to the left of the growing tile's path. There are two subcases depending on whether or not the right tile is the bottom-most tile in the left tile's right adjacency list. If it is, the right and growing tiles are made adjacent by just adding them to each other's adjacency lists. If not, the growing tile breaks into the middle of that right adjacency list, removing the right tile from that list and adding it to its own right adjacency list (see Figure 3B).

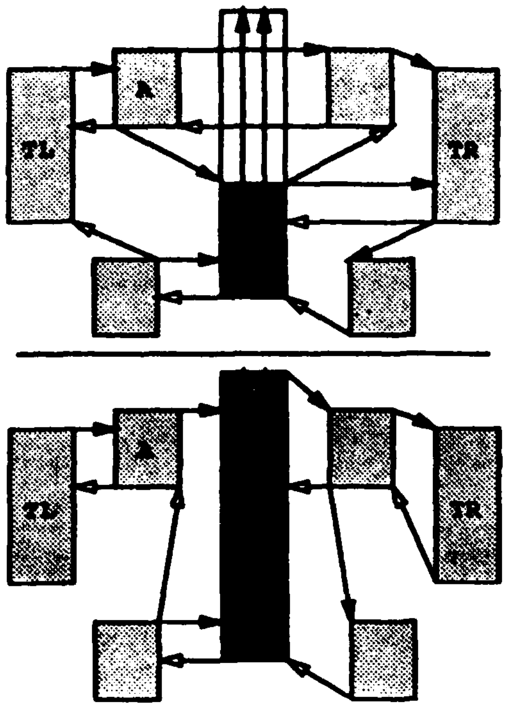


Figure 4: Before and After Growing a Tile

Sometimes when a tile is added to the top of a growing tile's adjacency list that tile shadows the previous top adjacent tile, causing the old tile to be removed from the list. This occurs on the right when the growing tile's up pointer is non-nil and on the left when a tile's down pointer points to the growing tile (tile A in Figure 4).

After the growing upwards loop is finished there may still be one more adjacency to find on each side of the grown tile. On the right side the top half of the last scanned turnaround is reversed scanned for a tile whose bottom is low enough to be adjacent to the growing tile (tile TR in Figure 5). A similar search-til-turnaround is made to the left for a new top left adjacent tile (TL).

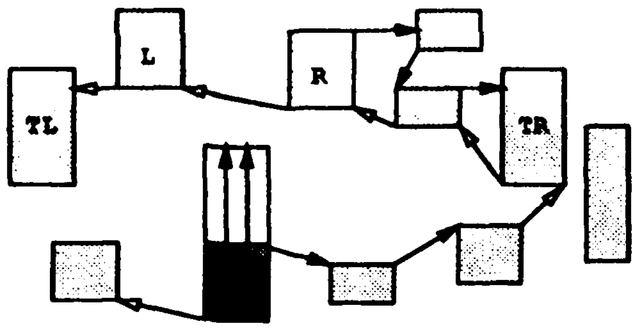


Figure 5: A Final Search for Top Adjacencies

Shrinking Tiles

Shrinking a tile is the reverse of growing one. We will just describe moving bottoms up since the top down case is symmetric. As a tile shrinks it may lose some of its adjacencies and cause new adjacencies to be stitched between tiles on either side of its old position. The algorithm is a loop, moving the bottom up, dropping one tile at a time from the shrinking tile's adjacency lists.

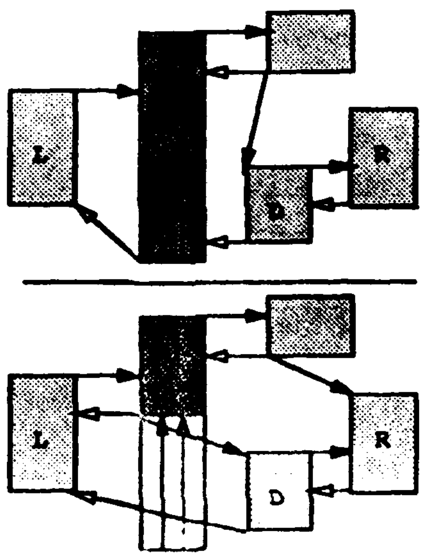


Figure 6: Before and After Shrinking a Tile

Two searches are done as each tile is dropped. When the bottom tile is removed from the right adjacency list a search is made to the right of that dropped tile to see if there is a tile that should be next in the shrinking tile's right adjacency list. The search follows right pointers until a turnaround is reached or a tile is found whose top is high enough to be adjacent to the shrinking tile's current bottom (tile R in Figure 6). A similar search is made to the left of the shrinking tile for a new top left adjacency (L) for the dropped tile. Removing the bottom left adjacency is symmetric, using left-up and right-down searches.

Moving, Deleting and Inserting Tiles

To move a tile a short distance it can be grown to strech over its new position and then shrunk to it's proper size. To move it a longer distance it is faster to delete it and then add it back, saving cutting and restitching many adjacencies.

To delete a tile it is first shrunk down to zero height so that there will be exactly one left and one right adjacent tile. The zero height tile is then removed from their adjacency lists and a simple check decides if the two tiles are now adjacent and need to be added to each other's adjacency lists.

Inserting a tile is the reverse of deleting one. A zero height tile is inserted between two tiles to start its adjacency lists and is then grown to its proper size. Locating these two tiles is a two step process. Starting at any tile (one cached to hopefully be nearby), up/right or down/left pointers are followed to find a tile at the same height as the bottom of the new tile. The up (down) pointers move quickly, but if one is nil or jumps over the desired height, the right (left) pointers also tend to go up (down) since they point to the tops (bottoms) of adjacency lists. In Figure 7, starting at tile S, one right and one up pointers are followed.

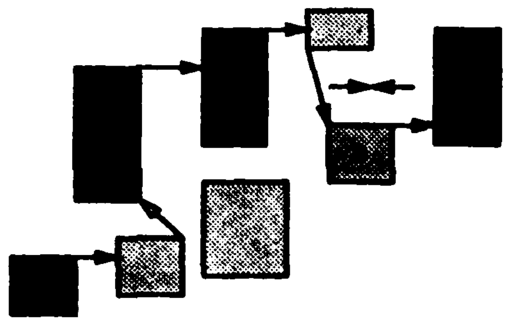


Figure 7: Locating Where to Insert a Tile

Once a tile is found at the correct height the adjacency lists are followed to the left or right to find pairs of tiles at the correct height until a pair is found with one tile on the left and one on the right of the new tile's position. The pairs of tiles need not be adjacent to each other; if in an adjacency list all the tiles are too high and/or low, the search is continued up and/or down from the tile(s) just below and/or above the wanted height, scanning until a turnaround. In Figure 7 the right tile is found by searching through a tile below the wanted height.

Quick Loading

The data structure of tiles could be loaded, using a given inital placement, by just inserting every tile. This would use $O(n^{1.5})$ time for roughly square layouts just locating where to insert all the tiles. Sorting the tiles lexicographically on the (y, x) coordinates of their lower-left-hand corners before inserting them makes the locates fast but it creates long horizontal frontiers which are sometimes searched as tiles are grown up to their proper heights, giving a faster but still $O(n^{1.5})$ time load.

It is much better to sort the tiles on (x, y). As they are inserted in this order each tile will have nothing to its right except the right edge of the frame. Thus a simple, quick load routine can be used. The locate reduces to finding the tile in the right edge's left adjacency list just below the new tile's bottom. The growing loop reduces to replacing some of the tiles in that left adjacency list with the new tile and using those tiles for the new tile's left adjacency list. The scan-til-turnaround checks still need to be made for a final top and bottom left adjacencies.

Note that this add routine is actually a fast, simple shadowing routine. The tiles in the right edge's left adjacency list correspond to the completely unshadowed tiles in the shadowing frontier. The partially shadowed tiles are found left or left-up from the tiles which shadow them.

Multiple Colors

So far we have had just one color (type) of tile, one spacing rule and one plane for tiles. For a real layout we separate the colors that do not interact into different planes. For nMOS there is a metal plane and a poly-diffusion plane. In a plane with two or more colors it is unlikely that using constant bloats for each color will satisfy all the spacing rules. Therefore the tiles are stored unbloated and when comparing two tiles their colors are used as indices into an array of spacing rules to find the appropriate bloat. The spacing distances are constrained to not bleed through tiles, that is, transitivity still holds; with constraints from P to Q and Q to R, none is needed from P to R. Rules with very large spacing distances (p- to n-diffusion in CMOS can bleed across poly) can violate this and need to be handled on a different plane.

Tile's tops and bottoms are now fuzzy since the effective bloat of a tile varies with the color of the tile with which it is being compared. This fuzziness causes two problems. The first is minor, occuring when a tile has zero (bloated) height as it is being inserted or deleted. Care must be taken that the original or final left and right adjacencies are valid because of possible vertical rule bleed-throughs.

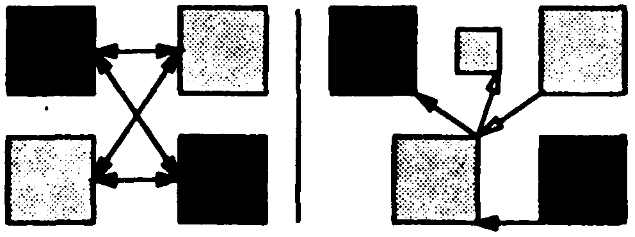


Figure 8: A. Crossed Constraints B. Up Pointer Conflict

The second problem, the around-the-corner problem, is more serious. Our data structure can only model planar graphs, that is what makes it possible to have just one up and one down pointer per record. But the fuzzy edges make it possible for two constraints to diagonally cross. A simple case is shown in Figure 8A where four tiles are arranged in a tight square, a diffusion tile above a poly tile on the left and below one on the right. The tiles can be close enough to generate four pairs of constraints since the poly-diffusion spacing rule is smaller than the poly-ploy and diffusion-diffusion spacings. Adding another tile in the middle of all this would make an up or down pointer need to point to two tiles at once (see Figure 8B).

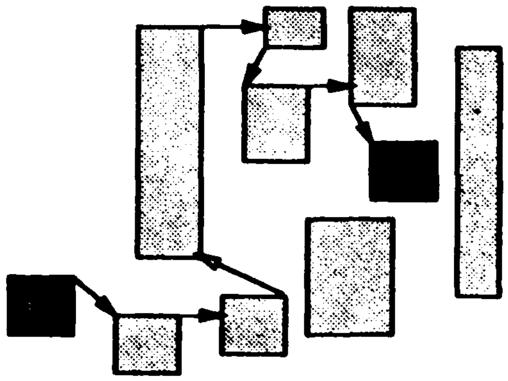


Figure 9: A Complex Around-the-Corner Search

Searches for around-the-corner adjacencies are now needed after growing and while shrinking tiles to avoid missing any constraints. In Figure 9 a search is made for an adjacency from the left dark tile. These constraints are kept in a seperate list since they cannot be included in the normal adjacency lists. This list is checked when the compaction constraints are generated to see if the constraints are still valid and, if so, if they can now be added to the normal lists. There is a simple test that eliminates or cuts short most searches by using the fact that a problem can only occur when tiles' top and bottom edges are very close. The test uses a precalculated function of the colors' bloat distances to determine the vertical extent of a search. This fudge distance is how far the top of a right tile must be above the top of a left tile before any tile high enough to be able to slide horizontally over the right tile will also be able to slide over the left tile.

Wires

So far we have discussed one kind of tile, but in layouts there two very different kinds of objects: wires and fixed-sized points. In our system, wires are connected to points, not other wires, at most one wire per side per point. A point is made at least as wide as the wire(s) connected to it to provide a full connection. When the point is wider, the wire has the freedom to slide along it. Wires could be represented as varying-sized tiles but we can take advantage of the fact that their ends are protected. A horizontal wire never generates any constraints needed for compaction in the x direction because it is always shadowed by the two endpoints to which it is attached. Thus, while points require two tile records, one for each dimension, we can get by with just one tile record per wire.

If we add two more pointers to each tile record, a wire-up and a wire-down, to doubly link wires and their endpoints together, we can take advantage of the fact that wires stretch when their endpoints move. Wires themselves never need to be moved; when an endpoint is shrunk away from its wire the wire grows to follow it. Thus instead of the usual shrink loop the routine can just see how many of the tiles in the endpoint's adjacency lists need to be moved to the wire's adjacency lists. Similarly for endpoints growing towards their wires. The wire-up and -down pointers also speed up the locates used for insertions and they can be used as turnarounds since nothing can pass between a wire and its endpoints.

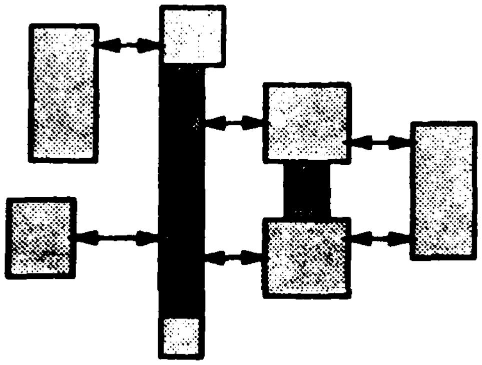


Figure 10: Wire Adjacencies

Since a vertical wire is horizontally constrainted by its endpoints, it follows, using transitivity, that if a tile is adjacent to one or both of a wire's endpoints it does not need to also be adjacent to the wire. Thus we can change the definition of adjacency for wires to say that a point and a wire are adjacent if and only if the point is adjacent only to the wire (on that side), and that two wires are never adjacent (see Figure 10). Note that now, unlike other tiles, a wire's left and right pointers can be nil and their up and down pointers are always nil.

It is relatively easy to modify the algorithms to handle this definition. A wire is allowed to temporarily become adjacent to a growing tile but it is removed before adding another tile to that side's adjacency list or, if needed, after the loop. A shrinking point becomes adjacent to a wire only when nothing else is available. While doing a locate and searching an adjacency list from a wire, if the list is empty or its tiles are all too high or all too low, the search is continued from the wire's endpoints.

The load routine is augmented with a load-wire routine which adds a wire as soon as both of its endpoints have been added. No locate is needed since the wire can be grown from one endpoint to the other. No final searches are needed either. The same simplifications are made for wire insertions. To delete a wire one endpoint is grown towards the other to zero out the wire. Then the wire is removed and the endpoint shrunk back.

Overlapping Tiles

In VLSI compaction most of the spacing rules can be relaxed between tiles that are electrically connected since there is no need to keep them from shorting out. Not allowing (bloated) tiles to overlap has the wasteful effect of forcing wires to have a minimum length because spacing rules keep their endpoints spaced apart.

Netlists and an array specifying which tile colors are allowed to overlap can be used to generate constraints allowing overlapping tiles. When a tile is moved up or down to overlap another tile the two tiles are forced to be adjacent to each other, to each other's left and right. Any wire between the two tiles is now unnecessary and is removed and added to a list of wires with negative length. After all the movements are done this list is checked to see if any of the wires now have positive lengths and should be added back to the graph.

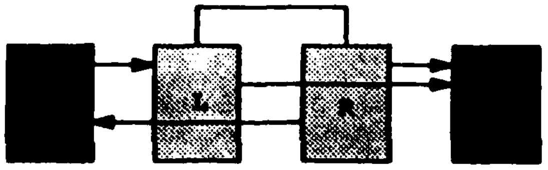


Figure 11: Constraints Using Netlists

A more general solution is to not generate any constraints at all between overlappable, electrically connected tiles, thus allowing them to pass through each other. This allows left and right jogs to switch and the ordering of taps off of buses to vary. Normally only one set of adjacency lists, say the right adjacencies, is needed to generate the constraints. When an adjacency between a left and a right tile is not used to generate a constraint, a routine is recursively called to find which additional tiles would be in the right adjacency list of the left tile if the right tile was deleted. A symmetric routine must also be used to find additional constraints to the left of the right tile. This has the effect of making overlappable, electrically connected tiles transparent to each other (see Figure 11).

While it is easy to generate constraints that allow tiles to pass through each other, when they actually do it can make updating the adjacency lists more difficult: pointers in both dimensions' data structure have to be adjusted, negative length wires can cause wire pointer conflicts and left- or right-of inconsistences can occur.

When a tile, moved up or down, passes through another tile, it creates a problem in the other dimension's data structure. The tile that was on the left is now on the right, a left adjacency pointer now points to a tile on the right and vis-a-versa. Deleting one of those tiles from that dimension's graph and reinserting it will fix the pointers.

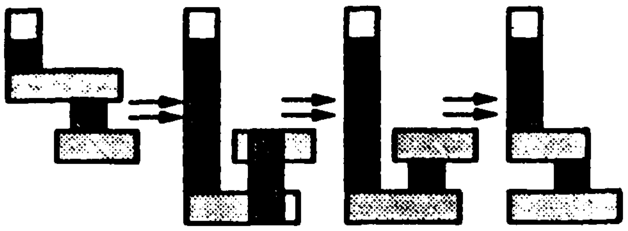


Figure 12: Fixing Negative Length Wires

If one endpoint of a wire passes completely through the other endpoint, the wire-up and -down pointers are swapped to give the wire a positive length. This can cause an endpoint's wire-up or -down pointer to need to point to two wires at once. The pointer pointing to the shorter wire is given preference. The longer wire is disconnected from that endpoint and attached to the other endpoint of the shorter wire (see Figure 12). This recurses until a free endpoint is found. A wire or endpoint may have to be moved horizontally slightly to make a full connection when a vertical wire's endpoints are changed.

Results and Conclusions

Currently this algorithm is programmed in C and uses the Lava compactor[1] to read in descriptions of sticks and do the actual compaction. The program quick loads both dimensions with an inital spread out solution and during the two one-dimensional compaction steps ends up moving all the tiles down and then to the left and updating both graphs. It takes about twice as long to move all the tiles as it does to do the quick load. Of course the updates are not really necessary for just two compaction steps. Using Lava's sorted every-pair constraint generation on the example in Figure 13 it takes, on a VAX 780, 5.7 cpu seconds to generate 19161 spacing constraints and 1 second to do the actual compaction. Using adjacency lists it takes 1.5 seconds to generate 847 spacing constraints (and update both dimension's data structures) and 0.18 seconds for the compaction.

We have an algorithm which handles multiple colors and planes, takes advantage of the special properties of wires and allows appropriate, electrically connected tiles to pass through each other. In our real-life examples there have been few around-the-corner constraints and pass-through problems. No more than an average of two spacing constraints per tile will ever be generated (that is how many adjacency pointers there are). The next step will be to use the data structure and algorithm in a simple down-and-left incremental compactor.

Our method combines the best of shadowing and corner-stitching. It is faster than shadowing, uses transitivity to generate the minimum number of spacing constraints and allows incremental changes without the overhead of recalculating all the constraints.

References

1. P.Eichenberger, *Fast Symbolic Layout Translation for Custom VLSI Integrated Circuits,* PhD dissertation, Stanford University, April 1986.
2. H.Baird, "Fast Algorithms for LSI Artwork Analysis", *Proceedings of the 14th Design Automation Conference*, ACM/IEEE, New Orleans, Louisiana, June 1977, pp. 303-311.
3. Christopher Kingsley, "A Hiererachical, Error-Tolerant Compactor", *Proceedings of the 21st Design Automation Conference,* ACM/IEEE, Albuquerque, New Mexico, June 1984, pp. 126-132.
4. Thomas Hedges, William Dawson and Y. Eric Cho, "Bitmap Graph Build Algorithm for Compaction", *Proceedings of the International Conference on Computer-Aided Design*, ACM/IEEE, Santa Clara, California, November 1985, pp. 340-342.
5. H.Watanabe, *IC Layout Generation & Compaction Using Mathematical Optimization*, PhD dissertation, University of Rochester, 1984.
6. J.Ousterhout, "Corner Stitching: A Data-Structuring Technique for VLSI Layout Tools", *IEEE Transactions on CAD*, Vol. CAD-3, No. 1, January, 1984.

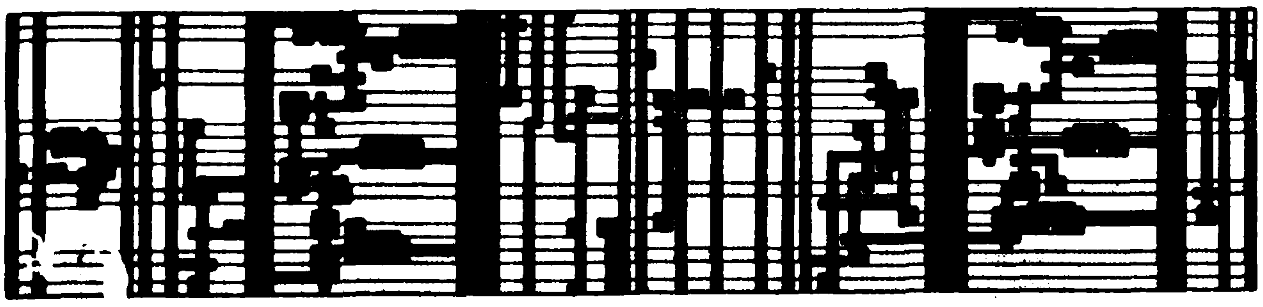


Figure 13: An Example With 587 Tiles, 282x64 lambda